

**U.S. NAVAL ACADEMY
COMPUTER SCIENCE DEPARTMENT
TECHNICAL REPORT**



High Energy Laser Progressive Wavefront Modeling

Izbicki, Michael J. Needham, Donald M. (Advisor)

USNA-CS-TR-2006-03

December 9, 2006

Report Documentation Page			Form Approved OMB No. 0704-0188		
Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.					
1. REPORT DATE 09 DEC 2006		2. REPORT TYPE		3. DATES COVERED 00-12-2006 to 00-12-2006	
4. TITLE AND SUBTITLE High Energy Laser Progressive Wavefront Modeling			5a. CONTRACT NUMBER		
			5b. GRANT NUMBER		
			5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S)			5d. PROJECT NUMBER		
			5e. TASK NUMBER		
			5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Naval Academy, Computer Science Department, 572M Holloway Rd Stop 9F, Annapolis, MD, 21403			8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)			10. SPONSOR/MONITOR'S ACRONYM(S)		
			11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited					
13. SUPPLEMENTARY NOTES The original document contains color images.					
14. ABSTRACT					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES 27	19a. NAME OF RESPONSIBLE PERSON
a. REPORT unclassified	b. ABSTRACT unclassified	c. THIS PAGE unclassified			

Computer Science Department
SI495A: Research Project Report
Fall AY07

High Energy Laser Progressive Wavefront Modeling

by

Midshipman M. J. Izbicki, 083306

United States Naval Academy
Annapolis, MD

Date

Certification of Faculty Mentor's Approval

Associate Professor Donald M. Needham
Department of Computer Science

Date

Department Chair Endorsement

Professor Kay G. Schulze
Chair, Department of Computer Science

Date

Abstract

High energy lasers have the potential to revolutionize naval warfighting by providing a weapons platform that has greater precision and speed than anything currently available. These lasers can be mounted on ships for surface warfare or mounted on satellites for strikes anywhere around the world. Crucial to the development of these lasers is an understanding of how different atmospheric conditions affect the laser's propagation and the shape of the beam when it finally illuminates the target. Dr. Bill Colson from the Naval Postgraduate School Physics Department developed a computer model for simulating these beams; however, his program can only output two dimensional slices of the three dimensional laser. Theoretically, the beams should be forming "noodles" of energy that break off from the main beam, but that can be difficult to see from Colson's original output. This project aims to modify Colson's program so that it can create three dimensional models of the laser beams, and show the progression of the beams over time.

Table of Contents

1. Introduction	3
2. Background	3
2.1 Compiling and Running Colson's Code	5
2.2 Analyzing Colson's Code	6
3. Rendering the 3D Simulation	7
3.1 Finding a Rendering Solution	7
3.2 The VolumeViz Render	8
3.2.1 Generating the Data	9
3.2.2 Culling the Input Data	9
3.2.3 Interpreting the Results	13
3.2.4 Generating Movies with VolumeViz	14
4. Analysis and Future Work	15
4.1 Analyzing the End Product	16
4.1 Analyzing Culling Algorithms	16
4.1 Scientific Analysis	17
5. Conclusions and Future Work	18
6. References	18

1. Introduction

Lasers have many potential applications for enhancing the effectiveness of the Navy, particularly in the realm of advanced weapons systems. Modeling how lasers propagate through the environment is an important part of developing these future weapons. When new lasers are developed, one of the first tests that is done is to study the beam's "burn pattern," that is the shape of the beam when it hits the target [3]. The focus of this project is to alter an existing algorithm for computing the propagation of a laser wavefront, making the output more useful. Dr. Bill Colson of the Naval Postgraduate School Physics Department developed a program that uses an algorithm he developed to generate a postscript file that describes a beam's characteristics. His output included three two-dimensional cross-sections of the laser: a top, side, and front view. While the front view provides a good description of how the laser would burn the target, the three views taken together provide little visual insight as to how the laser actually propagates through the atmosphere. Of particular interest is the possible formation of "noodles," or sections of laser that branch off from the main path and may or may not reconnect. This project aims to improve the utility of Colson's algorithm by providing an accurate three-dimensional rendering of the laser that will let the observer analyze any section of the laser's path.

The second section of this report provides background information on Colson's algorithm and focus on analyzing Colson's code. The code itself is written in poorly formatted C that no longer conforms to the languages standards and contains little in the way of explaining what it is actually doing. The third section examines techniques for rendering the laser in 3D, how it was implemented, and how movies are generated that show the laser's wavefront progressing as a function of time. The fourth section analyzes how effectively these modifications improve the viewers understanding of the laser propagation and the fifth section examines the possibilities for future work in this field. Instructions for installing and running the final product are in appendices D and E respectively.

2. Background

Colson's algorithm [1] works by inserting lenses into the path of the laser beam and propagating the beam according to the appropriate optical laws of physics. Each lens represents an effect of the physical environment on the beam. Examples include the effect of smoke and dust particles in the beam's path, wind, and even the effect of the laser heating the air as it passes through it. We will now examine the specifics of the output and analyze the strengths and weaknesses of the current presentation. The corresponding input file can be seen in Appendix A, and it's full output in Appendix B.

Conventions used in this document:

- When referring to the laser's dimensions, the x and y axes are defined as labeled on the x-y end view of Colson's original output, with origin at the center of the beam, and the z axis will be defined along the "length" of the beam with origin at the start.

- All commands and source code will be given in Courier New, example: make

Figures 1-3 are taken from the final output from Colson's code. The lighter areas represent the energy of the laser, and 95% of the laser's energy is contained within the white outline. The labels on the sides represent relative distances only, and are unit-less. Figure 1 shows the end view of the laser after passing through all the lenses. If a sensor were placed at the target end of the laser, this is what it would detect. From this view we can determine that there is a strong concentration of energy in the center of the beam, with smaller patches off to the side. Due to the rough nature of the output, however, we cannot determine exactly how these patches are related to each other. They could be areas where the main beam is weaker, or they could be the distinct "noodles."

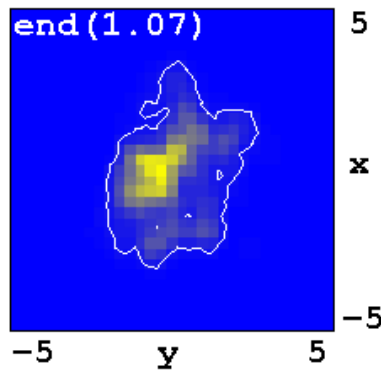


Figure 1: Sample Beam End View

Figure 2 shows the side view of the beam from Figure 1. This view gives a more detailed representation of how the beam propagates. The laser concentrates its energy in the center of the beam midway through the propagation. Small, distinct patches of energy demonstrate some evidence for the formation of noodles, but it is not clear how these might relate to the patches we saw in the end view.

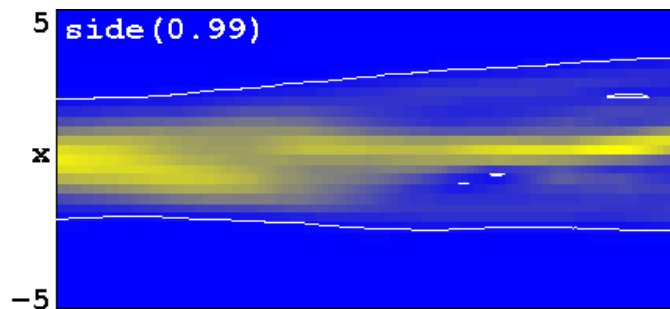


Figure 2: Sample Beam Side View

Figure 3 shows the top view of the beam. In this view beam's energy distribution remains relatively constant and concentrated throughout its entire propagation. Unlike the side view from Figure 2, there is no evidence for noodle creation, and little insight is provided into the beam's final end view. Although they represent the same beam propagation, these views provide conflicting information and are difficult to analyze. A

three dimensional model should help mesh these views into a more comprehensible representation and remove ambiguity.

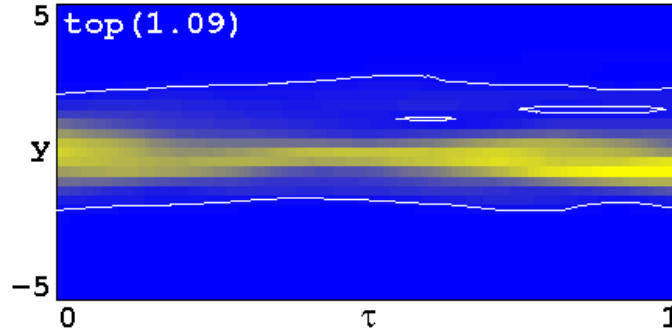


Figure 3: Sample Beam Top View

2.1. Compiling and Running Colson's Code

Colson's code comes packaged in a single file called Code.tar.gz. The extracted directory contains one very large source file called beamnew.c, a Makefile, and several example inputs. The machines in the Sun lab have gcc version 4.0.1 installed. Running make with this compiler will result in a series of errors. Appendix C.1 addresses how these errors were fixed. Once the program is compiled, running make should now generate the executable beamnew, which can be run by the command `./beamnew < input > output.ps`. Appendix A has an example input file, and Appendix B the resulting output. Figure 4 shows the dataflow pipeline for Colson's unmodified code.

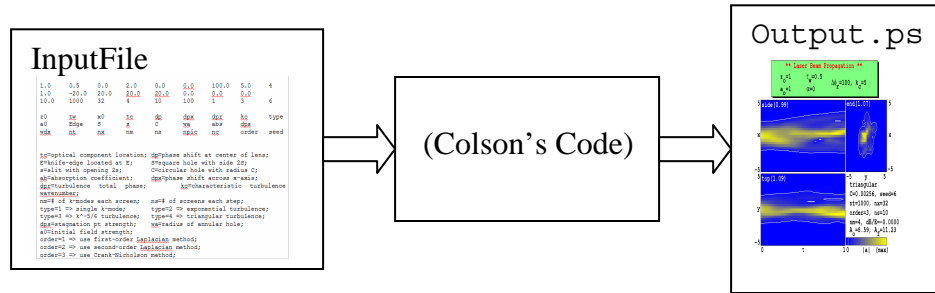


Figure 4: Dataflow pipeline for the Colson's unmodified code.

It is typically easier to work with several small source files than one large one, so the beamnew.c file was divided into four separate files: Greeks.c, cmath.c, beam.c, and math.h; a new src/beamnew directory was created to house these files. The beamnew.c file had comments clearly showing where each of these files began and ended inside of it, and appeared merely to be a merging of these formerly separate files. Greeks.c contains constant definitions, cmath.c math functions and postscript output functions, math.h prototypes for the cmath.c file, and beam.c contains the main function and all of the physics. The Makefile was then modified to reflect these changes. All further additions to Colson's code are encapsulated in pre-compiler commands of the form:


```

#ifdef _IZ_ADDITION_NAME_
    // new code
#endif

```

So that changes can be easily identified and removed by uncommenting its respective `#define` at the beginning of the file.

2.2. Analyzing Colson's Code

Colson's code was developed over time by many different people and is not easily understood without the relevant background physics knowledge. Variables are given one or two letter names that give little indication as to their intended use, and there is little whitespace separating the code into logical modules. The code is, however, separated into distinct segments by comments which give usually clear descriptions of what that section is doing.

The first step in the analysis began by looking for the code that rendered the graphics in the postscript output, hoping to be able to use this to figure out how the data was stored. Near the end of the file were three sections labeled "side view," "top view," and "end view," corresponding to the respective picture in the output. A summary of how the important variables for this process are used and created follows. Note that all double values fall in the range $[0,1]$.

- `int nt`: defines the number of discrete steps in the z axis (read from input)
- `int nx`: defines the number of discrete units in the x and y axes (read from input)
- `int itp, t`: The physics simulation calculates intensity values by starting the beam at the origin and "stepping" through each element in the z axis. `t` counts the current iteration through the loop and goes from 0 to `np` (read from the input file), and `itp` is a "culled" version of `t` used as a graphics index.
- `double ar[x][y], ai[x][y]`: real and imaginary components respectively of physics model at point (x, y)
- `double ayy[x][y]`: matrix representing intensity values in the end view. This matrix is created for every increment of `itp`, but for memory reasons overwrites whatever was in it previously. This means that what is in it at the end is what will be displayed in the end view.
- `double ax[z][x], ay[z][y]`: matrices representing intensity values in the side view (**ax**) and top view (**ay**). The matrix is updated every time `itp` is incremented according to the formulas:

$$ax[itp][i] = \sqrt{ar[i][no2]^2 + ai[i][no2]^2}$$

$$ay[itp][i] = \sqrt{ar[no2][i]^2 + ai[no2][i]^2}$$
 where `i` is an iterator for a loop over the range $[0, nx)$, and `no2` is the distance into the beam to take it's cross section of, set as default to the center (`no2=nx/2`).

3. Rendering the 3D Simulation

Rendering and modeling 3D objects is a well developed field of computer science with applications in many other fields of science. Advanced imaging techniques such as CAT scans are used by doctors to take images of patients' internal organs that require are most descriptively rendered in a 3D environment. More similar to this project, engineers involved in fluid dynamics simulations create algorithms to model the changes in density and flow of fluids as a result of placing an object in its path. As a result of the wide applicability of this problem, there are already many available applications designed to render various 3D problems. Dr. Colson's code was never designed to output a 3D rendering of the beam, however with minor modifications it can be changed to output data suitable for rendering in one of these 3D environments. New models for rendering 3D data, such as Schafhitzel's model [7], might provide more insight into the shape of the beam than a standard 3D representation, and Mehl's techniques might be applied to create a program that could classify the shapes found in the beam [5]. Building a simple 3D model is an appropriate first step towards a fuller analysis of the shape.

3.1. Finding a Rendering Solution

Because 3D rendering is so important, there are a large number of rendering solutions available, each with their own strengths. Most of the very advanced ones are proprietary. They cost large amounts of money to use and are available only on specific platforms. To take advantage of their unique features requires computing power only available to those with access to large clusters of machines or super computers. Rendering a laser beam with Colson's model proved to be relatively straightforward, so there was no need for these solutions. The alternative was to use open source solutions. These are attractive because they reduce the overall cost of the project but more importantly they make building off of this work easier for someone in the future. There are no restrictions to the distribution of the final product, and anyone who wants to continue work on this project will be able to unrestricted by copyright.

Two attractive open source solutions were the Vis5D+ program [2] and the VolumeViz program that comes with vtkFLTK [4]. Vis5D+ was originally designed to model weather patterns, and therefore includes the ability to use a time-dimension to display changes in the model. This could easily be used to render movies of the beam propagating through the model. Its input files, although well documented, are large and awkward to manipulate because of the extra functionality used to model weather patterns. At the recommendation of Dr. Stahl, the 3D rendering expert at the Naval Academy, we ended up using the VolumeViz program instead. It is actually an example program demonstrating how the Visual Toolkit (VTK) and the Fast Light Toolkit (FLTK) API's can be used to render 3D models [8]. It uses an intuitive and simple input file format and is easy to use. By using the well documented VTK, any changes in functionality needed would be easy to implement.

3.2.1. Generating the Data

In order to generate a 3D rendering of a laser beam, a program needs to create a three dimensional array of intensity values over the x, y, and z axes. Colson's code generates these values, but does not store them because the original code has no use for them. By storing each iteration of the axy array, however, we can easily generate them. Initially the idea was to have the beamnew program generate an intermediate, human readable, data file. This data file could then be processed by any number of programs to convert the data into a format suitable for rendering. This solution is very modular, allowing new rendering solutions to be added with ease. Appendix C.2 contains the code segment in beam.c right after it generates the next slice in the ax and ay matrices that generates the three dimensional data and stores it. Ultimately, the intermediate file created by the `_IZ_VIS5D_` segment became unnecessary overhead. Once one adequate rendering solution was found, there was no need for adding others. The `_IZ_VOLVIS_` segment which is used in the final product takes out this intermediate step and generates data directly readable by the VolumeViz program.

Because the VolumeViz program is relatively simple, its input doesn't require many extras and is straightforward. It takes input in the form of horizontal "slices" containing a two-dimensional x-y grid and then compiles these slices into a 3D model. It stores this data in the directory `out/`, and each slice is a file in the directory named `out.n`, where n is the z coordinate of that slice. The individual data points are stored as single bytes and contain values from 0-255. Each x-y grid is a 32x32 matrix of these data points, and there are exactly 100 horizontal slices. These values were chosen for reasons discussed in the Section 3.2.2 on culling and are hard coded into the program.

3.2.2. Culling the Input Data

Colson's code generates more data than is practical to handle. With each slice containing a 1024x1024 matrix of 4-byte doubles, at typically 1000 slices per image, nearly 4 gigabytes of information is produced for each image rendered. Furthermore, although less important, generation time scales proportionally with data output, and certain large inputs left the computer running all night without generating a usable output. Our focus was to reduce these space and time requirements while still getting data that accurately depicts the laser beam. The first step to solving this problem was to not store unused memory. Because each double in the matrix only stored a value from zero to one, we can compress it into a single byte. This is done by multiplying the value by 256 (the maximum size of one byte), and truncating the decimal places. Next, by aggressively reducing the number of slices and size of each matrix we can minimize our output. Empirically, by timing the program's running time with the `time` command and changing the input variables in the input file, we can determine which variables are causing the program to slow down the most. Following this method showed that the generation time scaled linearly with the `nt` variable and quadratically with the `nx` variable. This makes sense, because `nt` controls the number of slices along the time axis, and `nx` the width and height of those slices. See tables 1 and 2.

Colson's Code provides a partial solution to this problem. The output to the postscript file already uses a culled version of the number of slices. Every time the simulation passes through another time-unit, the variable t gets incremented up to a maximum of nt . Every tenth time the simulation passes through another time unit, the variable itp gets incremented. By moving the 3D output from the t loop into the itp loop, we store only one tenth the information we were storing before. Program run time remains mostly unaffected, however, because we still must perform the calculations for each iteration. Because these new slices are taken on a regular interval and spread throughout the dataset, they provide a slightly less accurate although significantly condensed version of the original output. We can justify that the new output is still accurate enough for the purposes of analyzing the beam's shape because although it does not use all the information Colson's Code generates, it uses all the information that would have been available for the postscript output. In the example input files, nt was always set to 1000, which would result in a culled length of 100. This length is hard-coded into the modified VolumeViz program; changing this value in the input file will result in only the first 100 slices being rendered, no matter how many exist. Figure 6 shows how this is done visually. Variable i indicates the index of the slice before culling and variable n indicates the index after culling. Darkened squares indicate slices that remain after culling. Each of these darkened squares is then further culled as in figure 7.

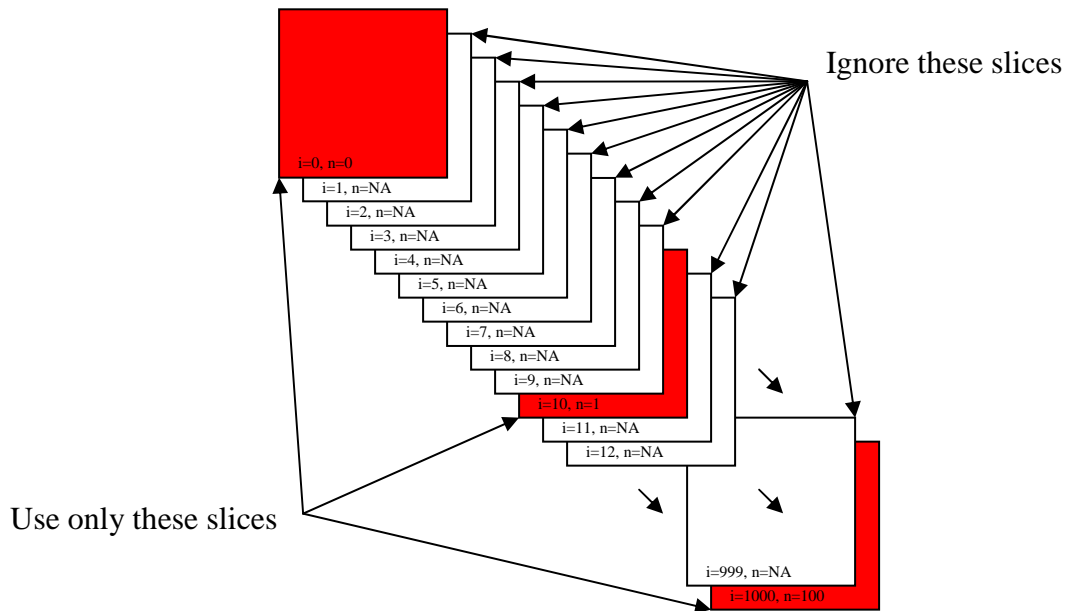


Figure 6: Culling the nt Variable

Culling the data corresponding to the nx variable proves much more challenging and more important. nx controls the width and the height of the time slices. Colson's original code has no method for culling this data, and memory and time grow quadratically with respect to this variable. The first step for finding an acceptable solution is to examine the output as this variable changes. It is impossible to do this directly, however, because of the way computers generate random numbers. Given an initial seed, the computer will generate a sequence of unpredictable numbers that are

always the same for that seed. It appears that all of the interesting looking beams, that is those that may be generating noodles that a 3D analysis would help with, are modeled in a way that they use random numbers to simulate things such as wind. By increasing the size of the x-y grid, we change the number of random numbers generated with each time interval, causing the program to be further along in the random sequence than it would have been with a smaller x-y grid, causing a fundamentally different beam shape. We could simply leave this how it is and require our program to accept only a certain size nx , like we did with nt , but unlike with nt this approach would limit the types of beams we would be able to analyze because nx equal to 64 will give a different beam than will nx equal to 32. Table 1 shows how the beam shape will change as a function of nx .

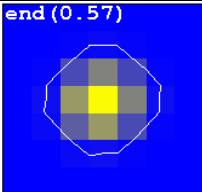
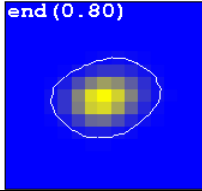
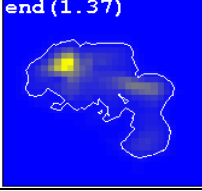
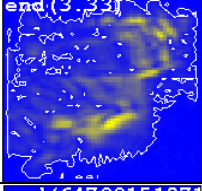
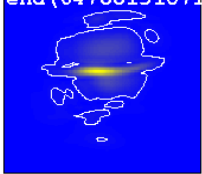
nx	Time (s)	Output size (_IZ_VIS5D_)	Output size (_IZ_VOLVIZ_)	End view from Colson's output
8	.5	0.5 MB	6.4 kB	
16	1	2.3 MB	25.6 kB	
32	4	9.2 MB	102 kB	
64	16	37 MB	410 kB	
128	64	148 MB	1.6 MB	

Table 1: Beam Shape and Properties a as a Function of nx

By culling all large inputs into the smallest one that maintains enough information to visually analyze, we can render all beams and reduce the amount of space required for previously unmanageable ones. Table 2 shows how the beam's shape and accuracy are

affected by culling. It may be argued that the beams where n_x is 32 may contain all interesting beam shapes, however, increasing the value of n_x will always increase the accuracy of the beam because it increases the number of data points that the physics simulation is run on. Even if those points are then subsequently downsized into a more manageable number, because they were calculated with more information their reduced version will remain more accurate. It is like rounding error when doing a math problem: you get more accurate results if you round at the end rather than after each step.

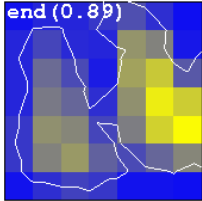
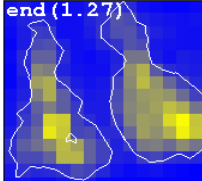
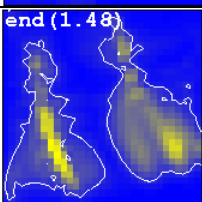
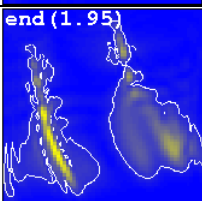
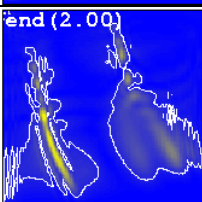
n_x	CULL_SIZE	Time (s)	Output size (_IZ_VIS5D_)	Output size (_IZ_VOLVIZ_)	End view from Colson's output
128	8	25	0.5 MB	6.4 kB	
128	16	26	2.3 MB	25.6 kB	
128	32	31	9.2 MB	102 kB	
128	64	35	37 MB	410 kB	
128	128	61	148 MB	1.6 MB	

Table 2: Culled Beam Shape and Properties with a Constant n_x

An easy way to perform the culling is to create “culled areas” equal in size to the input size divided by the output size. Take the first point of the culled area and discard the rest. Advantages to this technique are its simplicity, ease of implementation, and, because it is the same one used by Colson’s code to cull along the time axis, consistency. A final size of 32 was hard coded into the program; using Table 2 as a reference, this should be enough to accurately represent the data while at the same time keeping it as

small as possible. Due to the limitations of the implemented technique, the input size must be a power of two. Figure 7 shows how an nx value of 128 is culled down to an equivalent one of 32. By taking the input size 128 divided by the output size of 32, we get that each block should represent an area four units across. We then take the first data point in this area (darkened in the figure) as representative of the entire block, discarding the rest. Because output size grows quadratically with nx , this reduces it by a factor of 16. In the worst case scenario nx is equal to 1024, and following the above yields a 1024 fold reduction in size, converting a 4 GB file into a 4 MB file. Techniques for more advanced culling are discussed in Section 4.1.

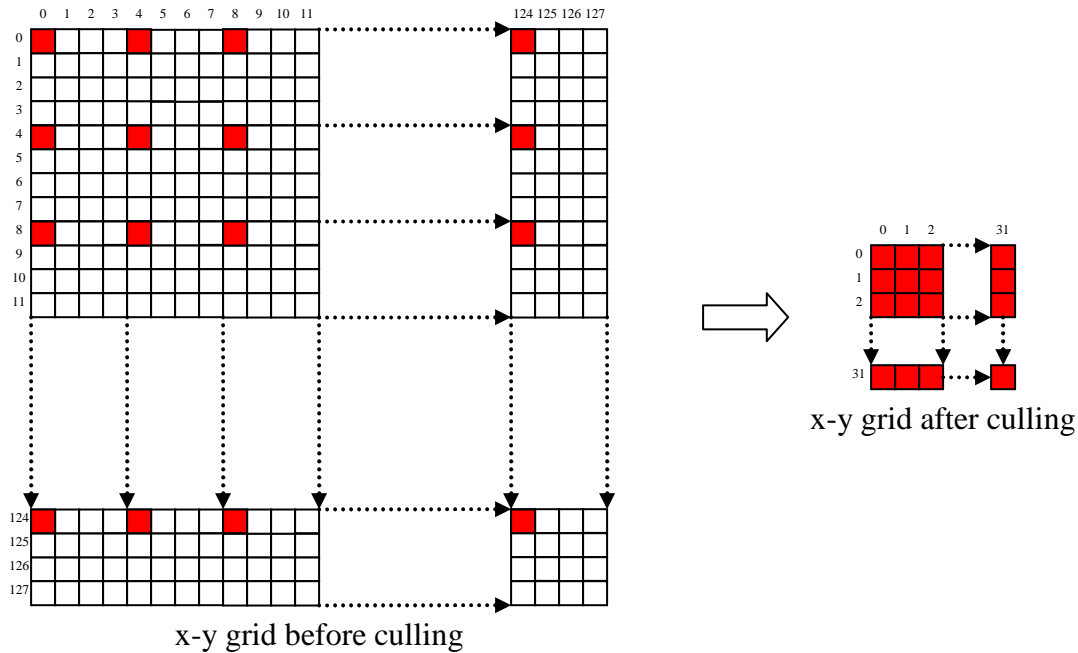


Figure 7: Culling the nx Variable

3.2.3. Interpreting the Results

The final images generated by the VolumeViz program represent the energy of the laser beam. There are two main views to select from: a contour view and a filled in view. The contour view generated by passing $-c$ to the program as in Appendix E. The contour is drawn wherever the beams energy is equal to 85 out of a maximum energy of 256. Therefore, all space contained within the contour has an energy of at least a third of the beam's maximum energy. The second method of rendering is by filling in the beam's space with colors and translucence according to the intensity at that point. This lets the user see all data points, but provides a less concrete picture of the edge of the beams energy. This is done by passing $-C$ to the program as in Appendix E. When no options are passed to the program, the image is generated as in Figure 9, with the left half rendered filled in, and the right half rendered as a contour.



Figure 8: Sample VolumeViz Output

3.2.4. Generating Movies with VolumeViz

Now that VolumeViz effectively renders static images of a beam, it should be relatively easy to string together a series of static shots to create a movie of the beam progressing along the screen. Movies have the advantage of showing the viewer a cross section of the entire beam, so that if anything interesting happens in the middle it will not be missed. One rudimentary way for generating the static shots is to simply change the number of slices that VolumeViz renders. The VolumeViz program was modified to change the number of slices it accepts using the `-g [numslices]` parameter, and a script created that iterates from volume `-g 0` to volume `-g 100`, creating 100 png files. The program `mencoder` can then be used to compile these images into an avi file, a standard format that will play in most video players. Figure 9 shows a representative selection of frames from a sample movie. In frame 1, the beam is still at its source and hasn't done anything interesting yet. By frame 40 we can see some turbulence develop in the core of the beam, and frame 100 shows the beam after it reaches the target. The energy in this beam remains relatively focused in the center.

This method successfully generates movies, but has some significant problems. The default position of the camera in the program is a top end view of the beam, so running the command `volume -g 0` generates a picture of the top of the beam. This angle gives little more insight into the beam's shape than the top view in Colson's original code, and doesn't take full advantage of the 3D. The program was therefore modified to accept camera angles on the command line. The `-r [angle]` rotates the camera right angle degrees, and the `-u [angle]` parameter rotates the camera angle up angle degrees. This introduced another problem, however. Certain angle combinations cause the program to crash when rendering certain stages of the beam progression (i.e. volume `-g 54 -r 15 -u`

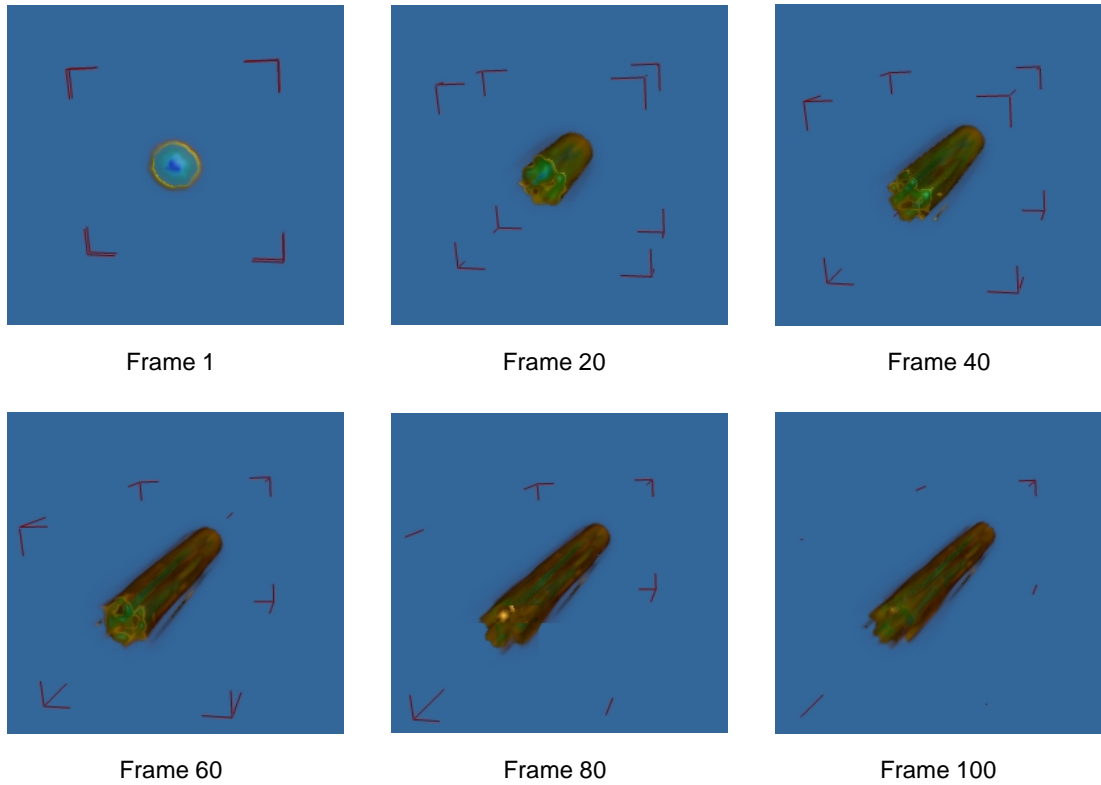


Figure 9: Wavefront Progression in Sample Movie

15 may crash, but volume `-g 60 -r 15 -u 15` and volume `-g 54 -r 30 -u 15` will not). This appears to happen about five percent of the time. Although the exact cause is unknown, it is not a result of bad input generated by the program. It appears to be a problem with the vtk library. The result is either a black png image or no output image. Calling the `run.sh` script to render movies would insert these black frames into the movie, interfering with the visualization. The script was therefore modified to keep track of which calls to volume fail and not include them in the final rendering, minimizing the effects of this problem. A second problem is that the VTK window will not render the static image unless it can create an X window to do so in. When invoking `run.sh`, this causes a series of windows to quickly pop up and close over the course of several minutes. Although the output of program remains unaffected by the issue, it does effectively prevent the computer from being used during generation, which is annoying.

4. Analysis and Future Work

The newly created rendering pipeline effectively lets the end user render 3D models of the lasers generated by Colson's code. These models allow the beams shape to be more easily and effectively analyzed; however, there are three main areas that can be improved upon: polishing the code by improving user's ability to interact with the improved functionality, improving the culling algorithms used in the code, and adding more scientific analysis features.

4.1. Analyzing the End Product

The interactive rendering of the beam is well implemented and works as designed, but the movie generation has several minor flaws. The way the program currently works, it renders each frame of the movie separately and then combines them into a movie. For an unknown reason, however, the program crashes with certain combinations of angles and frame indices, generating blank frames in the movie. A quick fix to this was to simply not include these blank frames in the final product. This creates noticeable jumps in the movie where several frames are skipped and is not an ideal solution. The proper fix for this problem would be to stop the program from ever crashing. The problem probably lies in the way the VolumeViz program interacts with the VTK library, but may be a bug in the library itself. Either way, this could be a very time consuming process. An easier solution will be presented next.

A second problem lies in the massive inefficiency of this method for generating movies. Every time a frame is generated, the VolumeViz program must regenerate the entire beam up to that point, despite the fact that it has already done it before. Thus, when rendering one hundred different frames for the movie, the image for the first frame gets rendered one hundred distinct times, and has other images concatenated on the end of it ninety-nine times. A form of memorization can eliminate this problem. Rather than rendering each frame individually, the VolumeViz program could render only the final frame, then gradually show less and less of it for each progressive frame. Because only one frame is rendered in this technique rather than 100 in the old technique, this has potential to dramatically improve render time. Implementing this would require looking through the VTK documentation to figure out how to manipulate what portions of the rendered image get displayed. This technique has the added bonus of fixing the crashing issue by completely circumventing it, as the program is now no longer asked to render half formed beams at various angles.

4.2. Analyzing Culling Algorithms

The biggest problem converting Colson's beam propagation data into 3D was the amount of information generated, up to 4 gigabytes. This made culling necessary, and the implementation used reduced the intermediate information down to several megabytes in the worst case. At this point, improving the culling algorithm is probably not necessary for size requirements, however it may be possible to increase the amount and accuracy of data stored in the small size. One easy way to improve the accuracy of the data is to use an averaging algorithm to shrink the data as opposed to the current method of picking the first data point. This has the disadvantage of being slightly more difficult to implement, having to keep track of averages through three dimensions, but has the advantage of considering every point in the final culled version.

There are two relatively easy algorithms that could increase the accuracy of the information contained in any given size of data. One way to measure the efficiency of the storage is to measure its entropy, or randomness. The theory behind this is that the more random a set of data is, the less room there is for an algorithm to compress it;

conversely, the less entropy in the data, the more room to compress it. There are many generic algorithms for compressing data. Huffman encoding, for example, takes advantage of the frequency of certain values throughout the data to more evenly distribute it. Ratanaworabhan's algorithm [6] was designed specifically for scientific computations so may be an effective drop in solution. Because we are getting the same sort of data every time, however, we can develop an algorithm specific to our purpose that may be more effective. Currently there is very little entropy in the culled data because most of the data points differ very little relative to the ones next to them. One method for taking advantage of this fact is to record the derivatives between individual points rather than the values of the points themselves. This takes advantage of the fact that most of the data points differ very little relative to the ones next to them, because while the changes in value may be very small, the changes in the derivatives may become quite large. Euler's method could then be used to integrate over these derivatives of the dataset to reconstruct the original [9]. These algorithms, while interesting to think about, would provide little in the way of practical improvements. At several megabytes for the largest datasets, the culled data is already small enough for any purpose, and if greater accuracy is desired, it is a simple matter of decreasing the culling amount.

4.3. Scientific Analysis

Ultimately, it is the goal of this project to give the user a better understanding of how different atmospheric effects are going to affect a laser's propagation. One way to do this would be to incorporate different physical models into the rendering pipeline. Thus if a scientist develops a propagation model that differs from Colson's, he could plug in his algorithm and see a 3D rendering of the beam it produces. This is actually relatively easy to do. The VolumeViz program will render images of anything that can produce intermediate data in the format it expects, i.e. a 3D grid of bytes representing intensity values of the beam at that point. To incorporate a new algorithm into the process, it simply needs to be modified such that it outputs its beam shape in that format. Adding this functionality to Colson's code was relatively easy, and adding it to another algorithm should be similarly easy. The new algorithm would probably need similar culling done to scale the data down to size.

The most promising area for further investigating beam propagation involves Monte Carlo simulations. A Monte Carlo simulation is done on algorithms that have a random component, and involves repeating the simulation many times and combining all of the results to see what it would probably look like if it were done again. Colson's code would be very suitable to this type of simulation. In the input file, the rand variable seeds the random number generator used in the simulation and has a huge effect on many of the propagations. In real life, however, you cannot simply choose which set of random variables nature will apply on your laser. By running the simulation many times, we can average each of the results to get a "what it probably would like" and take the standard deviation to find "how likely it would look like this." These results would probably be the most useful in determining if a laser could effectively hit its target given certain weather conditions.

5. Conclusions

Accurately modeling laser propagation is a critical step towards the development and application of laser based information and weapons systems. Dr. Colson developed a program for modeling a laser's propagation with a two dimensional output, but due to the limitations of the display format, a lot of information is lost and the beam shape is hard to discern. By connecting the algorithm from Colson's code to the VolumeViz program, we were able to effectively visualize the laser's propagation in a 3D environment. The algorithm developed by Colson generated enormous amounts of data, however, which had to be reduced to make generation and render time acceptable. Raw, the output could be as large as four gigabytes. A system of culling was therefore implemented which maintained the integrity of the data and managed to reduce the size down to several megabytes. The successful conversion of the beam's data into a 3D format provides additional insight into the beam's propagation that might have otherwise been lost.

6. References

1. *Laser Propagation Software*, Dr. Colson, Naval Postgraduate School.
2. S. Johnson, J. Edwards. *Vis5D+*. <http://vis5d.sourceforge.net/>.
3. A.H. Lumpkin, R.B. Fiorito, D.W. Rule, D.H. Dowell, W.C. Sellyey, A.R. Lowery, *Initial Optical-Transition Radiation Measurements of the Electron Beam for the Boening Free-Electron-Laser Experiment*, Free Electron Laser Conference, 1989.
4. S. McInerly. *The vtkFLTK bridge Library*. <http://vtkfltk.sourceforge.net/>.
5. A. Mehl, B. Blanz, T. Vetter, H.P. Seidel, *A statistical method for robust 3D surface reconstruction from sparse data*, 3D Data Processing, Visualization and Transmission, 2004.
6. P. Ratanaworabhan, J. Ke, M. Murtscher, *Fast Lossless Compression of Scientific Floating-Point Data*, Data Compression Conference, 2006.
7. T. Schafhitzel, D. Weiskopf, T. Ertl, *Interactive exploration of unsteady 3D flow with linked 2D/3D texture advection*, Coordinated and Multiple Views in Exploratory Visualization, 2005.
8. *The Visualization Toolkit*. <http://www.vtk.org/>.
9. Weisstein. "Euler Forward Method." *MathWorld*.
<http://mathworld.wolfram.com/EulerForwardMethod.html>
10. Weisstein. "Monte Carlo Method." *MathWorld*.
<http://mathworld.wolfram.com/MonteCarloMethod.html>

Appendix A

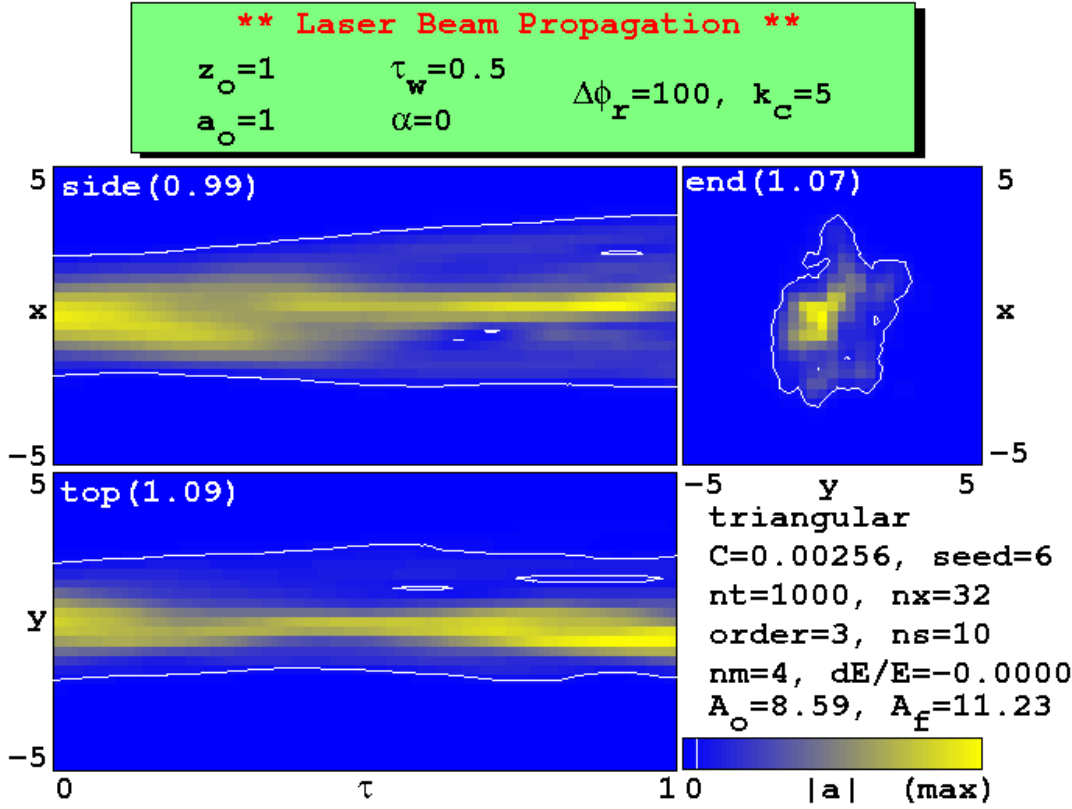
inbeamnew0: sample input file

1.0	0.5	0.0	2.0	0.0	0.0	100.0	5.0	4
1.0	-20.0	20.0	20.0	20.0	0.0	0.0	0.0	
10.0	1000	32	4	10	100	1	3	6
z0	tw	x0	tc	dp	dpx	dpr	kc	type
a0	Edge	S	s	C	wa	abs	dps	
wdx	nt	nx	nm	ns	npic	nc	order	seed

tc=optical component location; dp=phase shift at center of lens;
 E=knife-edge located at E; S=square hole with side 2S;
 s=slit with opening 2s; C=circular hole with radius C;
 ab=absorption coefficient; dpx=phase shift across x-axis;
 dpr=turbulence total phase; kc=characteristic turbulence
 wavenumber;
 nm=# of k-modes each screen; ns=# of screens each step;
 type=1 => single k-mode; type=2 => exponential turbulence;
 type=3 => $k^{-5/6}$ turbulence; type=4 => triangular turbulence;
 dps=stagnation pt strength; wa=radius of annular hole;
 a0=initial field strength;
 order=1 => use first-order Laplacian method;
 order=2 => use second-order Laplacian method;
 order=3 => use Crank-Nicholson method;

Appendix B

inbeamnew0.ps: Colson's original output from inbeamnew0 (Appendix A) rendered using gimp



Appendix C.1: Modifying Colson's Code

Fixing Compilation Errors

Colson's code came packaged in a single file called `Code.tar.gz`. The extracted directory contains one very large source file called `beamnew.c`, a `Makefile`, and several example inputs. The machines in the Sun lab have `gcc` version 4.0.1 installed. Running `make` with this compiler will result in a series of errors. These are caused by string literals with newlines in them, something allowed on many older compilers. To fix this problem, open `beamnew.c` in `emacs` and append `\n\` to the end of each occurrence. The `\n` maintains the formatting that the output expects, and the trailing backslash tells the compiler to treat the next line as though it were appended onto the current line. Running `make` should now generate the executable `beamnew`, which can be run by the command `./beamnew < input > output.ps`.

Appendix C.2: Modifying Colson's Code

Code segment of beam.c that outputs the 3D array of intensity values to a file

```

/* file dump */

#ifdef _IZ_VIS5D_
    for (x=0;x<nx;++x) {
        for (y=0;y<ny;++y) {
            a2=ar[x][y]*ar[x][y]+ai[x][y]*ai[x][y];
            if (a2>amx*amx) amx=sqrt(a2);
            Ef+=a2*dx2;
            ayx[y][x]=sqrt(a2);
            if (ayx[y][x]>ayxmx) ayxmx=ayx[y][x];
            fprintf(fp,"%f ",ayx[y][x]);
        }
        fprintf(fp,"\n");
    }

    fprintf(fp,"-----\n");
#endif

#ifdef _IZ_VOLVIS_
    char buff[40];
    memset(buff,0,sizeof(buff));
    sprintf(buff,"out/out.%i",itp);
    FILE* fp=fopen(buff,"w+");
    for (x=0;x<nx;++x) {
        for (y=0;y<ny;++y) {
            a2=ar[x][y]*ar[x][y]+ai[x][y]*ai[x][y];
            if (a2>amx*amx) amx=sqrt(a2);
            Ef+=a2*dx2;
            ayx[y][x]=sqrt(a2);
            if (ayx[y][x]>ayxmx) ayxmx=ayx[y][x];
            unsigned char ucp=(unsigned char)(ayx[y][x]*256);
            fwrite(&ucp,1,1,fp);
        }
    }
    fclose(fp);
#endif

```

Appendix D: Installation

Instructions for installing the laser rendering project on a unix system

1. Create a new directory `laser`, and download the following files:

- `proj.tar`
- `MPlayer-1.0pre8.tar.gz`
- `VTK-4.4.2.zip`
- `fltk-1.1.7-source.zip`
- `cmake-2.4.3-SunOS-sparc.tar.gz` (or the appropriate one for you architecture)

2. To build `mplayer`:

```
> gunzip MPlayer-1.0pre8.tar.gz
> tar -xf MPlayer-1.0pre8.tar
> cd MPlayer-1.0pre8
> ./configure
> ./make
> cd ..
```

3. To build `cmake`:

```
> gunzip cmake-2.4.3-SunOS-sparc.tar.gz
> tar -xf cmake-2.4.3-SunOS-sparc.tar
> cd cmake-2.4.3-SunOS-sparc
> ./configure
> ./make
> cd ..
```

4. To build `VTK`:

```
> unzip VTK-4.4.2.zip
> cd VTK
> rm Utilities/png/png.h    *** (Distribution includes obsolete file)
> ../cmake-2.4.3-SunOS-sparc/bin/cmake .
> ./make
> cd ..
```

5. To build `FLTK`:

```
> unzip fltk-1.1.7-source.zip
> cd fltk-1.1.7
> ../cmake-2.4.3-SunOS-sparc/bin/cmake .
> ./make
> cd ..
```

6. To install project files:

```
> tar -xf proj.tar
> cd proj
> ./make
> cd ..
> ./run.sh
```

Appendix E: Running the program

Instructions for generating the various types of output.

1. Using Colson's original program

Generate the postscript output file:

```
> ./beamnew < [inputFile] > [output.ps]
```

View postscript file:

```
> gs [output.ps]
```

2. Using the VolumeViz renderer

To facilitate use of the renderer, the script run.sh can be used to generate all the intermediate data.

- Perform all steps to generate a single movie from an input file

```
> ./run.sh -a [inputFile] [outputFile.avi] [options]
```
- Load an input file. This is required for all of the following options to work.

```
> ./run.sh -b [inputFile]
```
- Launch the interactive renderer

```
> ./Volume [options]
```
- Generate still images

```
> ./run.sh -p [options]
```
- Convert still images into avi

```
> ./run.sh -v [outputFile.avi]
```
- Generate still images and make avi in single step

```
> ./run.sh -vp [outputFile.avi] [options]
```
- Generate nine movies from loaded input that showcase the laser's properties

```
> ./run.sh -super
```

There are three options for rendering the shape of the beam. Passing `-c` renders the outside surface of the beam only, `-C` renders the contours inside the beam, and passing `-cC` uses both rendering techniques. Passing nothing causes the left half of the beam to be rendered as if `-c` were passed and the right side to be rendered as if `-C` were passed.

The default angle of the beam is looking at the destination end of the beam, offset by 15 degrees up and to the right. The `-r [angle]` and `-u [angle]` parameters can be passed to specify different angles.

